Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1–10
DOI: 10.15439/2017F001
ISSN 2300-5963 ACSIS, Vol. 11

Formal Definition of a General Ontology Pattern Language using a Graph Grammar

Eduardo Zambon
Federal University of Espírito Santo (UFES), Brazil
zambon@inf.ufes.br

Giancarlo Guizzardi
Free University of Bozen-Bolzano, Italy &
Ontology and Conceptual Modeling Research Group (NEMO),
Federal University of Espírito Santo (UFES), Brazil
giancarlo.guizzardi@unibz.it

***Abstract*—In recent years, there has been a growing interest in the use of ontological theories in the philosophical sense (Foundational Ontologies) to analyze and (re)design conceptual modeling languages. This paper is about an ontologically well-founded conceptual modeling language in this tradition, termed OntoUML. This language embeds a number of *ontological patterns* that reflect the micro-theories comprising a particular foundational ontology named UFO. We here (re)define OntoUML as a formal graph grammar and demonstrate how the models of this language can be constructed by the combined application of ontological patterns following a number of graph transformation rules. As a result, we obtain a version of this language fully defined as a formal *Ontology Pattern Grammar*. In other words, this paper presents a formal definition of OntoUML that is both explicit in terms of the ontological patterns that it incorporates and is completely independent of the UML meta-model.**

I. Introduction

IN RECENT years, there has been a growing interest in the use of ontological theories in the philosophical sense (Foundational Ontologies) and engineering tools derived from these theories to improve the theory and practice of Information Systems Engineering (ISE). In particular, there is a stable tradition on the use of foundational ontologies to analyze and (re) design conceptual modeling languages that play an essential role in ISE. For example, in [1], the authors have conducted an empirical study with 528 practitioners and have shown that the perception of ontological deficiencies in conceptual modeling languages negatively affects the perception of the usability and usefulness of these languages.

This paper is written in the context of a research program involving a particular Foundational Ontology, namely, the Unified Foundational Ontology (UFO) [2] and a particular conceptual modeling language derived from it, namely, OntoUML [3]. OntoUML was conceived as an ontologically well-founded version of the UML 2.0 fragment of class diagrams. Both UFO and OntoUML have gained increasing attention in the context of ontology-driven conceptual modeling. For example, a recent study shows that UFO is the second-most used foundational ontology in conceptual modeling and the one with the fastest adoption rate [4]. Moreover, the study also shows OntoUML is among the most used languages in ontology-driven conceptual modeling (together with UML, (E)ER, OWL and BPMN).

In a recent paper [5], we have shown that OntoUML comprises a number of *ontology patterns* reflecting corresponding ontological micro-theories put forth by its underlying foundational ontology (UFO) [6]. As discussed in [5], UFO is a system of micro-theories addressing basically all the classic conceptual modeling concepts. For each of the ontological distinctions present in UFO and which are reflected as modeling constructs in OntoUML, we have a corresponding axiomatization. This axiomatization makes sure that OntoUML constructs can only appear in a model forming clusters of constructs with their ties and associated constraints. In other words, in general purpose languages such as ER, UML or OWL, the actual modeling building blocks of the language are low-granularity modeling primitives such as class, association, attribute, etc. In OntoUML, in contrast, the actual modeling primitives are these structures (and their corresponding axiomatization) reflecting the underlying ontological micro-theories. As a consequence, OntoUML could be conceived as a *pattern grammar (language)* whose models are constructed via the combined instantiation of the ontological patterns.

In [5], we presented the ontological patterns embedded in OntoUML, the connection between these patterns, and their possible combination rules. However, the characterization of OntoUML as a full-blown pattern grammar was done there in an informal way. In this paper we remedy this situation by defining and implementing OntoUML as an **Ontology Pattern Grammar**. As the main contribution of this paper, we show how OntoUML patterns can be formally defined using a graph grammar based on the Single-Pushout Graph Transformation theory. Furthermore, we present a practical implementation of this grammar, using the general-purpose graph transformation tool GROOVE [7][8].

We highlight that the definition and implementation of OntoUML as a formal Ontology Pattern Grammar can bring several benefits to the (Ontology-Driven) Conceptual Modeling community, namely: (i) the grammar is defined in a formal, Turing powerful, computational method that circumvents the limitations of the current meta-modeling approaches for defining the abstract syntax of modeling languages; (ii) the language is defined in a way that affords its independence from the UML meta-model and, as consequence, the results presented here can be ported to other conceptual modeling languages (*e.g.*, some ontological distinctions put forth by UFO have been incorporated in the ORM language [9][10]) and employed by the conceptual modeling community at large beyond UML users; (iii) the language makes explicit its constituting ontology design patterns which, once more, reflect the ontological micro-theories put forth by UFO. In other words, in comparison to the current definition of OntoUML's abstract syntax (in terms of a UML 2.0 meta-model with associated OCL constraints), the implementation of this language in the manner proposed here affords a much higher *ontological transparency* for the language, *i.e.*, the implementation makes much more transparent the ontological commitments embedded in that conceptual modeling language. Finally, we highlight that the implementation of these patterns in a computational tool supports the construction of OntoUML models by employing modeling primitives of a higher-granularity (the ontological patterns). Moreover, since these higher-granularity modeling elements can only be combined to each other in a restricted set of ways, in each modeling step, the design space is reduced. We believe that this strategy reduces the complexity of the modeling process, especially for novice modelers.

The remainder of this paper is organized as follows. Sections II and III present the background of this work. In particular, Section III briefly introduces the basic concepts of graph transformation, including the commonly used Single-Pushout approach, and the definition of a graph grammar and its associated graph language. Section IV presents the syntactical conventions of the GROOVE tool set. Section V presents the definition of OntoUML as an Ontology Pattern (Graph) Grammar and shows its implementation in GROOVE. Finally, by using this implementation, in Section VI we illustrate the use of the proposed grammar to instantiate real OntoUML models. Section VII presents our final considerations.

II. UFO AND ONTOUML

OntoUML, as all structural conceptual modeling languages (*e.g.*, UML, ER, ORM), is meant to represent type-level structures whose instances are endurants, *i.e.*, they are meant to model **Endurant Universals** and their type-level relations.

Fig. 3 depicts the **Endurant Universals** hierarchy in UFO. A basic formal relation that can hold between (endurant) universals in UFO is the relation of subtyping. If a universal B is a subtype of a universal A then we have that: (i) it is necessarily the case that all instances of B are instances of A; and (ii) all properties of universal A are in a sense *inherited* by universal B, *i.e.*, Bs are As and, therefore, have all properties that are properties defined for universal A.

Endurant universals are distinguished into **Substantial Universals** and **Moment Universals**. Naturally, these are kinds of universals whose instances are **Substantials** and **Moments** [3], respectively. Substantials are *existentially independent* objects such as John Lennon, the Moon, an organization, a car, a dog. Substantials can have a mereologically complex structure, *i.e.*, they can have parts that are themselves substantials. In case these substantials are *functional complexes*, their parts are functional parts termed components (*e.g.*, a CPU is a functional component of a computer); in case they are *collectives*, they have a uniform structure in which all parts (termed members) are undifferentiated w.r.t. the whole (*e.g.*, in the sense all trees are considered merely as members of a forest) [3]. Moments, in contrast, are *existentially dependent* individuals such as John's headache (which depends on him) and the marriage between John and Yoko (which depends on both John and Yoko). Being existentially dependent entities, moments can only exist by *inhering in* other endurants [3].

Concerning the substantial universal hierarchy, **Sortal Universals** are the ones that either provide or carry a uniform *principle of identity* for their instances. A principle of identity supports the judgment whether two individuals are the same, *i.e.*, in which circumstances the identity relation holds. In particular, it also informs which changes an individual can undergo without changing its identity. Within the category of **Sortal Universals**, we have the distinction between rigid and anti-rigid universals. A rigid universal is one that classifies its instances necessarily (in the modal sense), *i.e.*, the instances of that universal cannot cease to be so without ceasing to exist. Anti-rigidity, in contrast, characterizes a universal whose instances can move in and out of its extension without altering their identity. For instance, contrast the rigid universal *Person* with the anti-rigid universals *Student* or *Husband*. While the same individual John never ceases to be an instance of *Person*, he can move in and out of the extension of *Student* or *Husband*, depending on whether he enrolls in/finishes college or marries/divorces, respectively. **Kinds** are sortal rigid universals that provide a uniform principle of identity for their instances (*e.g.*, *Person*). **Subkinds** are sortal rigid universals that carry the *principle of identity* supplied by a unique **Kind** (*e.g.*, a kind *Person* can have the subkinds *Man* and *Woman* that carry the principle of identity provided by *Person*). Concerning anti-rigid sortals, we have the distinction between roles and phases. **Phases** are relationally independent universals defined as a partition of a sortal. This partition is derived based on an intrinsic property of that universal (*e.g.*, *Child* is a phase of *Person*, instantiated by instances of persons who are less than 12 years old). **Roles** are relationally dependent (or *externally dependent*) universals, capturing relational properties shared by instances of a given kind, *i.e.*, putting it baldly: entities play roles when related to other entities via the so-called *material relations* (*e.g.*, in the way some plays the role *Husband* when connected via the material relation of "*being married to*" with someone playing the role of *Wife*). Since the principle of identity is provided by a unique **Kind**, each sortal hierarchy has a unique **Kind** at the top [3].

The relational dependence of **Roles** is manifested by the presence of a **Relator** (a particular type of moment that is existentially dependent on multiple individuals) in the model. **Relators** are individuals with the power of connecting entities. For example, an *Enrollment* relator connects a *Student* role with an **Educational Institution**. OntoUML has a construct for modeling relator universals. Every instance of a relator universal is existentially dependent on at least two distinct entities. The formal relation that take place between a relator universal and the object classes it connects is termed *mediation* (a particular type of *existential dependence* relation) [3].

Non-Sortals or **Mixins** are universals that aggregate properties that are common to different sortals, *i.e.*, that ultimately classify entities that are of different **Kinds**. Non-sortals do not provide a uniform principle of identity for their instances; instead, they just classify things that share common properties but which obey different principles of identity. *Furniture* is an example of non-sortal (a category) that aggregates properties of *Table*, *Chair* and so on. Other examples include works of art (including paintings, music compositions, statues), insurable items (including works of arts, buildings, cars, body parts, etc.) and social and legal objects (including people, organizations, contracts, legislations, etc.). The meta-properties of rigidity and anti-rigidity can also be applied to distinguish different types of **Non-Sortals (Mixins)**. A **Category** represents a rigid and relationally independent mixin, *i.e.*, a dispersive universal that aggregates essential properties that are common to different rigid sortals [3] (*e.g.*, *Physical Object* aggregates essential properties of *Table*, *Car*, *Glass*, etc). A **RoleMixin** represents an anti-rigid and externally dependent non-sortal, *i.e.*, a dispersive universal that aggregates properties that are common to different **Roles** (*e.g.*, a *Customer* that aggregates properties of *Individual Customer* and *Corporate Customer*) [3].

The leaf ontological distinctions represented in Fig. 3 as well as their corresponding axiomatization (*i.e.*, their corresponding ontological micro-theories) are reflected as modeling constructs in OntoUML [3]. Moreover, as shown in [5], this axiomatization ensures that the OntoUML constructs representing these ontological categories can only appear in a model forming clusters of constructs with their ties and associated constraints. In other words, as previously mentioned, the actual modeling primitives of OntoUML are certain *pattern-based structures* reflecting the ontological micro-theories comprising UFO. Thus, OntoUML is a pattern language whose models are constructed via the combined instantiation of certain foundational patterns. As a pattern grammar, an OntoUML model is a non-empty set of **Endurant Universal Expressions**. These expressions, in turn, as summarized in Table I, are defined in a recursive manner reflecting the taxonomic structure of the UFO ontology of Endurant Universals (Fig. 3) until a level of concrete terminal elements (kinds and concrete ontology patterns) is reached. The OntoUML patterns have already been presented in [5] but at a more informal level. In this paper, we present the OntoUML patterns in a formal manner, using a graph transformation system.

TABLE I
EXPRESSIONS OF ONTOUML.

Expression	Expression Structure
Endurant Universal Expression	Substantial Universal Expression or Moment Universal Expression
Substantial Universal Expression	Sortal Expression or Mixin Expression
Sortal Expression	Rigid Sortal Expression or Anti-Rigid Sortal Expression
Rigid Sortal Expression	Substance Sortal Expression or SUBKIND PATTERN
Substance Sortal Expression	<<kind>> T or COLLECTIVE PATTERN
Anti-Rigid Sortal Expression	PHASE PATTERN or ROLE PATTERN
Mixin Expression	CATEGORY PATTERN or ROLEMIXIN PATTERN
Moment Universal Expression	MODE PATTERN or RELATOR PATTERN
Relationally Dependent Universal Expression	ROLE PATTERN or ROLEMIXIN PATTERN

III. Graph Transformation

A. Basic Concepts

Graph transformation (or *graph rewriting*) [11] has been advocated as a flexible formalism, suitable for modeling systems with dynamic configurations or states. This flexibility is achieved by the fact that the underlying data structure, that of graphs, is capable of capturing a broad variety of systems. Some areas where graph transformation is being applied include the visual modeling of systems, the formal specification of model transformations, and the definition of graph languages, to name a few [12][8].

Essentially, whenever a system consists of entities with relations between them, this can be naturally captured by a graph in which nodes stand for entities and edges for relations. If, in addition, a main characteristic of such a system is that entities are created or deleted and the relations between them can change, then the transformation of graphs comes into play.

The core concept of graph transformation is the rule-based modification of graphs, where each application of a rule leads to a graph transformation step. A transformation rule specifies both the necessary preconditions for its application and the rule effect (modifications) on a *host graph*. The modified graph produced by a rule application is the result of the transformation.

In this work, we use graph transformations to formally model the construction of ontology patterns. A set of graph transformation rules can be seen as a declarative specification of how the construction of an ontology model can evolve from an initial state, represented by an initial host graph. This combination of a rule set plus an initial graph is called a *graph grammar*, and the (possibly infinite) set of graphs reachable from the initial graph constitute the grammar *language*.

In its basic form, our formal graphs are composed of nodes and directed labeled binary edges. Fig. 1(a) shows a graph representing a single-linked list composed of five cells (nodes labeled C) and a sentinel node (L) to mark the head and tail elements of the list. Labels C and L are actually part of self-loop edges; however, for visual convenience, unary labels are written inside their associated node and the edge is omitted. Node identities are displayed at the top left corner of each node (edge identities are not shown). Edges labeled n indicate

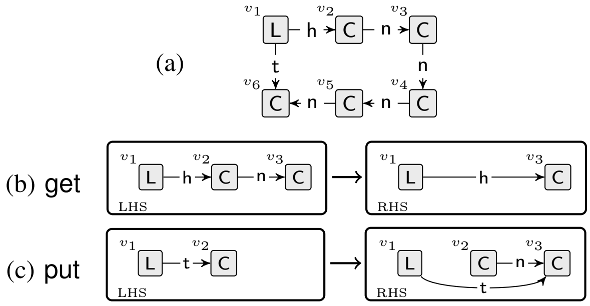


Fig. 1. (a) A graph representing a single-linked list with five elements. (b),(c) Two graph transformation rules.

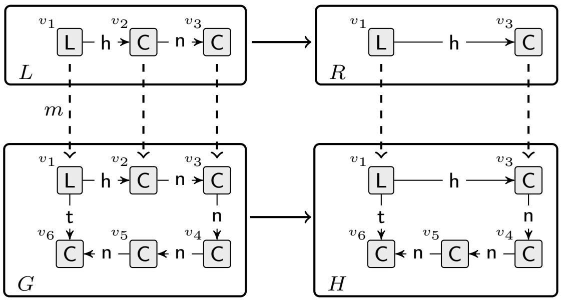


Fig. 2. Example of a graph transformation, with application of rule get to the graph of Fig. 1(a). Match m is indicated with dashed arrows.

the *next* list element, and labels h and t, indicate the list *head* and *tail*, respectively.

Graphs are modified according to transformation (or production) rules, that describe both the conditions for their application and the changes that should be performed to the host graph. In its basic form, a *transformation rule* r is composed of two graphs, a left-hand side (LHS) L and a right-hand side (RHS) R. Fig. 1(b,c) shows rules for removing the head element of a list (get) and inserting a new element at the tail of the list (put). For rule get, we have that the set of deleted nodes is $\{v_2\}$, indicating that the head cell is removed by the rule. Additionally, set $\{\langle v_1, \mathsf{h}, v_2\rangle, \langle v_2, \mathsf{n}, v_3\rangle\}$ corresponds to the set of edges to be removed. Rule get does not create new nodes, and set of edges to be created consists solely of $\{\langle v_1, \mathsf{h}, v_3\rangle\}$. For the put rule given in Fig. 1(c) these sets can be analogously inferred.

Graphs are related by *morphisms*, structure preserving functions over nodes and edges that also respect edge labels. For a rule r to be *applicable* to a host graph G, a *match* m of L into G has to exist, where m must be structure-preserving, *i.e.*, m is a morphism from L to G. The *application* of r to G according to match m comprises two steps. First, all nodes and edges matched by $L \setminus R$ are removed from G. In the second step of rule application, elements of $R \setminus L$ are added to G, to obtain the derived graph H. Fig. 2 depicts the application of rule get (Fig. 1(b)) to the host graph of Fig. 1(a), under match m. The commuting square of morphisms corresponds to a *pushout* in Category Theory, therefore this type of construction for graph transformation is dubbed the Single-Pushout (SPO) approach.

By associating an initial host graph to a set of related rules we obtain a *graph grammar*. A graph grammar defines a *graph language*, the set of all graphs reachable from the initial host graph. If a grammar has at least one rule that is always enabled (*i.e.*, that has an empty LHS), then the grammar language is infinite. However, a finite fragment of a language can still be algorithmically generated. This is the core functionality of the GROOVE tool set, which calls this action *exploration of the grammar state space*. We describe the GROOVE tool in Section IV.

A graph grammar is a Type 0 grammar according to the Chomsky Hierarchy and therefore graph transformations can be seen as an alternative, Turing powerful, computational method [13]. However, despite their theoretical power, graph grammars still require further extensions to be applicable in practice. In this work, we use the concepts of *typed graph grammars* and of *rule schemata*, described in the following two sections.

B. Node Types and Inheritance

A typed graph transformation with node type inheritance [14] is a formalization of the inheritance concept common to object oriented (OO) systems. The core concept of this formalization requires enriching a graph grammar with a (transitive) inheritance relation over node types. Using the usual graph transformation terminology, the inheritance relation is described by a *type graph* (roughly equivalent to a class diagram, in OO terms) that describes all valid structure of rule and host graph elements.

Roughly speaking, a graph grammar can be typed according to a type graph $\mathcal{T}$ by the construction of a morphism from any grammar graph (rule or host graphs) into $\mathcal{T}$. If no such *typing morphism* can be constructed, then the grammar is considered erroneous. Tools such as GROOVE are properly equipped to handle type graphs and node inheritance, and give error messages if a grammar cannot be typed.

Although we refrain ourselves from presenting the theory of typed graph transformation due to its complexity (an interested reader is referred to [14]), in practice the consequences of using types in a graph grammar are quite straightforward, affecting only the rule matching mechanism. For example, suppose two node types **S** and **T**, with **S** a subtype of **T**. Any occurrence of a **T**-node in the LHS of a rule can be matched by either a **T**- or **S**-node in the host graph. This idea can be generalized to a complete transitive inheritance relation and is properly implemented in the GROOVE tool [8].

Fig. 3 depicts a type graph that describes a UFO-A fragment of Endurant Universals, as presented in [5] and discussed in Section II. The type graph is able to properly capture the complete hierarchy as given in [5], with the exception of annotations such as *disjoint* and *complete*. In addition, the type graph in Fig. 3 can be seen as a formal representation of the recursive relationships between expression structures, as informally presented in Table I of Section II. In Table I, for example, a **Anti-Rigid Sortal Expression** can stand for either a **Phase Pattern** or a **Role Pattern**. This is formalized

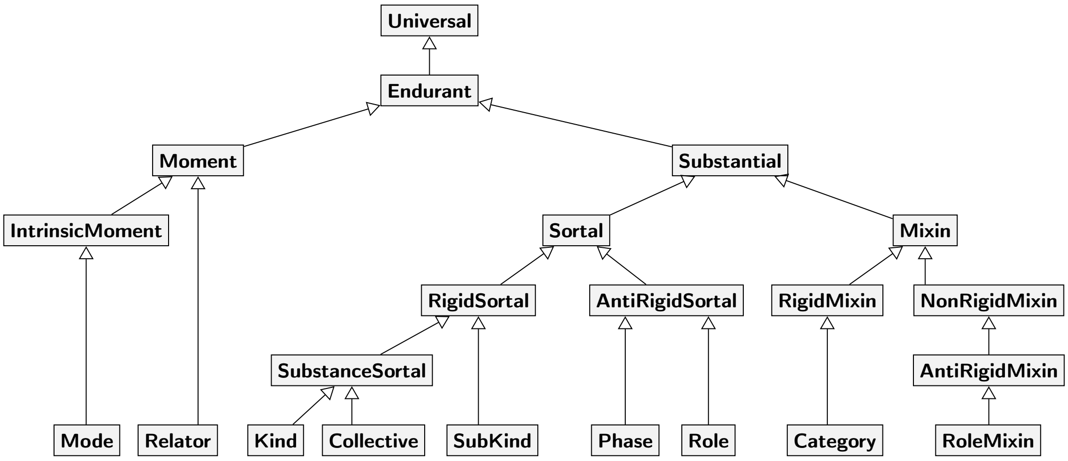


Fig. 3. Type graph describing a UFO-A fragment of Endurant Universals.

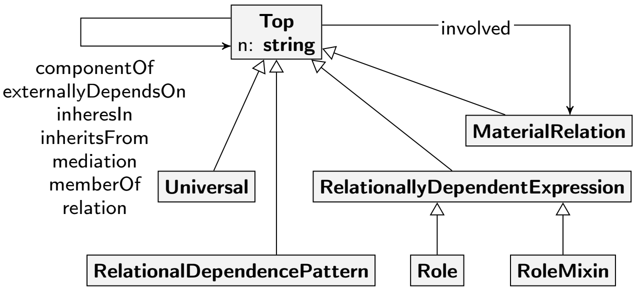


Fig. 4. Additional type graph used in the Ontology Pattern Grammar.

in the type graph by the subtyping (inheritance) relations between node types **AntiRigidSortal**, **Phase**, and **Role**. It is also important to note that some types in Fig. 3 have been abbreviated for convenience. For example, an Endurant Universal is summarized as the **Endurant** node type.

Not all expressions given in Table I are part of the UFO-A fragment of Endurant Universals. Therefore, to completely and formally define the graph structures allowed in the Ontology Pattern Grammar, we also use the additional type graph elements shown in Fig. 4. For convenience, we introduce the most general node type **Top**, and unify all edge types allowed using this node type. The string attribute n within the **Top**-node is used to name the constructs as they are introduced in the ontology graph. Also worthy of note is node type **RelationalDependencePattern**, whose purpose is to serve as a place-holder, representing an intermediate construction on the ontology graph that will later be replaced by further rule applications (see Section V).

From here on, we assume that all graph grammar elements presented are typed by the type graph $\mathcal{T}$ that is formed by the merge (on equal node types) of the type graphs shown in Figs. 3 and 4.

C. Rule Schemata

If a node type **T** is not a leaf in the type graph $\mathcal{T}$, then any rule where a **T**-node occurs in the LHS actually describes a "collection of rules", where the elements of this collection are formed by instantiating the **T**-node with all its subtypes. Although this is not necessary in practice, conceptually speaking, a transformation rule that uses node type inheritance can be seen as a *rule schema*, which defines a family of *concrete* rules by means of the inheritance relation.

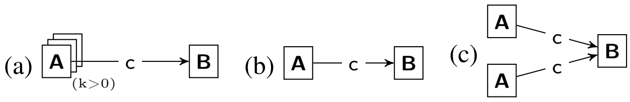


Fig. 5. (a) LHS of a rule schema describing an arbitrary number of **A**-nodes connected to a single **B**-node. (b),(c) Instantiation of such schema for $k = 1$ and $k = 2$, respectively.

This idea of rule schemata or *rule collections* can also be used capture the concepts of element *multiplicity* and quantification [15][16]. Fig. 5(a) shows a rule schema to handle node multiplicities. Node **A** is depicted with extra copies to indicate that it can represent an arbitrary number of concrete nodes, with such number bound by parameter k. Fig. 5(b,c) shows the concrete schema instantiation for $k = 1$ and $k = 2$, respectively. Again, for simplicity's sake, we will not elaborate further on the theory of rule schemata, referring the reader to [15] for details.

A rule schema for node types describe a *finite* collection of rules, since any type graph $\mathcal{T}$ must be finite. However, in a rule schema for multiplicities this is usually not the case: parameter k can be unbounded, leading to an arbitrarily large collection of concrete rules. This may pose a problem, since the definition of a graph grammar requires the set of transformation rules to be finite. Nevertheless, this problem can be easily fixed by imposing an upper bound n to k, thus limiting the number of concrete rules under consideration. The value for n is dependent on the application scenario of the graph grammar; in most practical scenarios (including the Ontology Pattern Grammar), n is a small (single digit) natural. In Section V, for each grammar rule that required the use of a schema, we specify the value of k used upon instantiation.

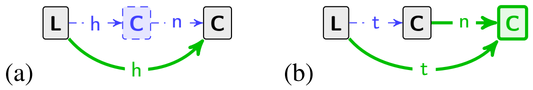


Fig. 6. Rules (a) get and (b) put in GROOVE notation.

IV. GROOVE

GROOVE [7][8] is a general purpose graph transformation tool set that uses directed labeled graphs. The core functionality of GROOVE is to (partially) compute the language of a graph grammar, by recursively applying all rules from the grammar to the initial host graph, and to all graphs generated by such applications. In the tool terminology, this exploration results in a *state space* consisting of the generated graphs. The main component of the GROOVE tool set is the Simulator, a graphical tool that integrates (among others) the functionalities of rule and host graph editing, and of interactive or automatic state space exploration.

A graph transformation rule is composed of two graphs L and R, as defined previously. However, in practice, it is tedious and rather repetitive to describe a rule in terms of its composing graphs. Therefore, in GROOVE, both L and R are combined into a single graph, and colors and line strokes are used to visually distinguish them. Fig. 6 shows the get and put rules previously given in Fig. 1(b,c), now in GROOVE notation. The semantics of this notation is summarized as follows:

- The black (continuous thin) components are *reader* elements, which must be present during matching and are preserved by the rule application.
- The blue (dashed thin) components are *eraser* elements, which must be present during matching and are deleted by the rule application.
- The green (continuous fat) components are *creator* elements and are created by the rule application.

V. OntoUML as a Graph Grammar

In this section we describe the main contribution of this paper, namely the **Ontology Pattern Grammar**. In [5], we discussed at length the *static* structure of OntoUML patterns, focusing mainly on the rationale for the usage of a pattern, but without concern with the actual sequencing of pattern constructions that may lead to a complete model. On the other hand, in Section III, we described the major concepts of graph transformation, a formalism aimed at specifying the *dynamic* evolution of graph structures. In this section we merge these two concepts.

Our goal is to use graph transformations to formally capture the dynamic evolution of an OntoUML model from its inception until its final form. To do so, we specify each step in the construction of a Ontology Pattern as the application of a graph transformation rule. This level of granularity in the model construction is justified by the fact that, in OntoUML, the patterns are the actual modeling primitives, as previously stated.

Tables II and III show all graph transformation rules that form the Ontology Pattern Grammar, as implemented in GROOVE. The initial host graph is empty and thus it is not depicted. Certain patterns admit two or more variants, which are presented consecutively in Tables II and III. Additionally, rules whose names end in ki are based in rule schemata, with i indicating the concrete value used in the schema instantiation. In these cases, we indicate in the rule description which nodes are multiple (*i.e.*, have an associated $k > 0$).

With exception of the **Kind Pattern** rule, for any other rule to be applicable, an existing structure must already be present in the model (these are the *reader* and *eraser* elements in the rules). Also, every rule creates an additional graph structure (*creator* elements) with each application. Thus, by sequencing a series of rule applications, the ontology model (which starts empty) grows until reaching its final form, with the GROOVE tool ensuring that only valid (applicable) transformations can be taken at each step. Therefore, the final model created is guaranteed to be structurally and ontologically sound *by construction*.

The first two cells of Table II show the rules for creating a **Category Pattern**, which has two variants. Variant 1 creates a **Category** node for an existing **Mixin** node to inherit from. Variant 2 comes from a schema, where the **RigidSortal** node is multiple, with the rule instance using $k = 2$. Thus, in this rule, a category serves as the inheritance point of two rigid sortals. The **Collective Pattern** also comes from a schema, with the **Endurant** node being multiple (the rule instance uses $k = 1$). Thus, in this rule, a new **Collective** node is created as a member of a single existing endurant. The **Component**, **Inheritance** and **Membership Pattern** rules are used to respectively create the relations of parthood, inheritance and membership among two existing endurants.

The **Kind Pattern** rule is used to create new **Kind** nodes. This is always possible, since a kind has no preconditions to be introduced in the model. Therefore, the **Kind Pattern** rule is always applied first in a new graph (model). The **Mode Pattern** rule follows a schema, with the **Endurant** node the mode depends on being multiple. Here, this multiple node is instantiated with $k = 1$. A similar construction occurs in the **Phase Pattern**, with the multiple **Phase** nodes instantiated for $k = 2$, indicating that two distinct phases can inherit from an existing sortal.

The **Relational Dependence Pattern** has three variants to handle distinct structures of mediation by a **Relator** node. In Variant 1, the mediation is direct, whereas in Variants 2 and 3 the mediation occurs through the membership of a substantial. In either case, the goal of these rules is to confirm the existence of a relator mediating a **Relationally Dependent Expression** (either a **Role** or **RoleMixin** node). The rules then erase the temporary **RelationalDependencePattern** marker node which was previously created when a **Role** or **RoleMixin Pattern** was introduced.

The **Relator Pattern** has two variants, with Variant 1 handling the creation of a **Relator** node that directly mediates one or more substantials. Table III shows instances of the rule schema for $k = 1, 2, 3$. Variant 2 behaves similarly but also introduces a reified **MaterialRelation** node to connect

TABLE II
ONTOLOGY PATTERN GRAMMAR IN GROOVE (PART I)

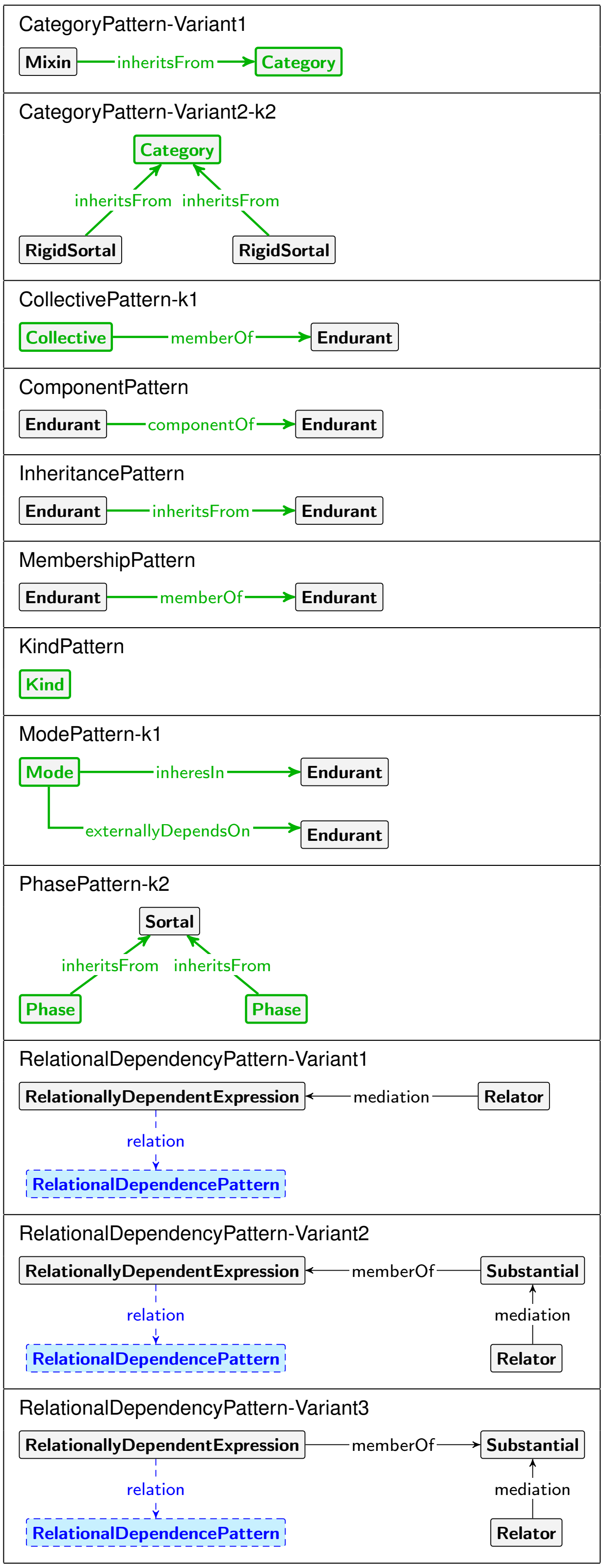
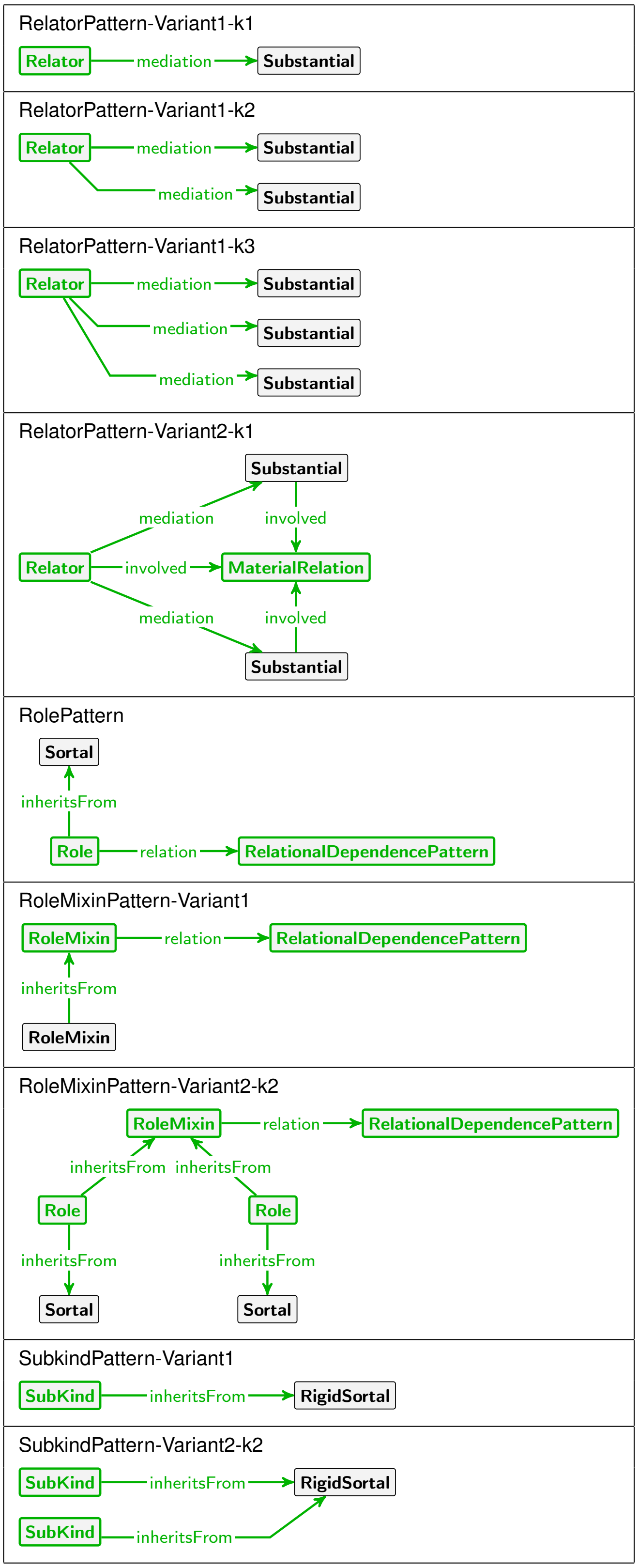


TABLE III
ONTOLOGY PATTERN GRAMMAR IN GROOVE (PART II)



the involved substantials. This reification is necessary because GROOVE only admits edges connecting nodes (not other edges).

The **Role Pattern** rule creates a **Role** node that inherits from an existing sortal. A role is a **Relationally Dependent Expression** which must be connected to a **Relator Pattern**. Since the relator can only be created after all mediated substantials exist, the rule creates the **RelationalDependencePattern** marker node, to indicate that there is a unresolved dependence in the model. Subsequently, after one or more **Relator Patterns** are created, this marker node is removed by a **Relational Dependence Pattern** rule. A model with one or more **RelationalDependencePattern** nodes is at an intermediate state of construction, and cannot be considered finished until all dependencies are satisfied.

The **Role Mixin Pattern** is similar to the **Role Pattern**, with the distinction that this pattern creates a **RoleMixin** node to aggregate one or more roles. Variant 2 instantiates the rule schema with $k = 2$, both for the **Role** and **Sortal** nodes. Variant 1, on the other hand, allows the introduction a mixin that generalizes an existing one. Finally, the last two cells of Table III show the rules that creates the **Subkind Pattern**.

The main functionality of the GROOVE tool is state space exploration (language enumeration) of a graph grammar. Although this functionality can be used with the Ontology Pattern Grammar to (partially) enumerate consistent OntoUML models, this is not the grammar intended use, as the exploration can quickly exhaust computational resources. Conversely, the Ontology Pattern Grammar was designed to be used with the interactive mode of GROOVE, where the user (modeler) decides at each step which rule to apply to introduce a new pattern. This sequencing of rule applications is illustrated in the next section with two examples.

VI. Applying the Ontology Pattern Grammar

In order to illustrate the application of the Ontology Pattern Grammar to produce OntoUML models, we use two existing published models of [5]. The versions of these models produced using the grammar are shown in Figs. 7 and 8, respectively. In the model of Fig. 7, we see on the top-left side the result of an application of a **Kind Pattern** (*Person*) followed by an application of the **Phase Pattern** (*Deceased Person* and *Living Person* specializing the sortal *Person*). In the top-right side of the model, we see an analogous application of the same patterns creating the kind *Organization* and the phases *Extinct Organization* and *Active Organization*. In the center of the model, we see the application of the **RoleMixin Pattern** Variant 2 creating the rolemixin *Customer* and the roles *Personal Customer* and *Corporate Customer* that specialize the sortals *Living Person* and *Active Organization*, respectively. The **Relational Dependent Pattern** node (let us call it RDP-1) created by this **RoleMixing Pattern** Variant 2 serves as a marker to indicate that the rolemixin *Customer* requires a relation with a pattern that is still not present. Continuing with the model construction, the role *Supplier* is introduced via an application of the **Role Pattern**. This role also requires a **Relational Dependent Pattern** (call it RDP-2). Finally, a relationally dependent expression is introduced with a relator (*Purchase Contract*) and the material relation *purchases from* via the application of the **Relator Pattern** Variant 2. Once the relator *Purchase Contract* is created, both RDP-1 and RDP-2 are satisfied and thus their temporary node markers can be removed with two applications of the **Relational Dependency Pattern** Variant 1. This concludes the model creation, yielding the graph shown in Fig. 7.

In the model of Fig. 8, we can start with the applications of the **Kind Pattern** creating the kinds *Organization*, *Organizational Unit* and *Person*. After that, with the application of the **Component Pattern**, we can make an *Organizational Unit* a component of an *Organization* (both *Organizational Unit* and *Organization* are **Endurant** types). We have then two applications of the **SubKind Pattern** Variant 1 creating the subkinds *Car Rental Branch* and *Car Rental Agency*. Again these two types can be connected by an application of the **Component Pattern**. An application of the **RoleMixin Pattern** Variant 2 creates the rolemixin *Car Rental Provider* as well as the roles *Car Rental Branch Provider* (that specializes the sortal *Car Rental Branch*) and *Car Rental Agency Provider* (that specializes the sortal *Car Rental Agency*). The **Relational Dependency Pattern** instance (let us call it RDP-1) associated to this pattern is left unresolved at this moment. We can then apply the **RoleMixin Pattern** Variant 1 to create the rolemixin *Service Provider* as a supertype of *Car Rental Provider*, leaving a second instance of the **Relational Dependency Pattern** unresolved (let us call it RDP-2). We can apply once more an instance of the **RoleMixin Pattern** Variant 2 creating the rolemixin *Potential Car Renter* and the roles *Potential Person Car Renter* (specializing the sortal *Person*) and *Potential Organization Car Renter* (specializing the sortal *Organization*). Again, we leave an instance of the **Relational Dependency Pattern** unresolved (RDP-3). We can apply again the **RoleMixin Pattern** Variant 1 creating the rolemixin *Target Customer* as a supertype of *Potential Car Renter*, leaving a final instance of a **Relational Dependency Pattern** unresolved (RDP-4). Then, using the **Collective Pattern** we can introduce the *Target Customer Community* as a member of the endurant *Target Customer*. This collective then appears as the supertype of the *Potential Car Renter Community* subkind, created via a **Subkind Pattern** Variant 1. Rule **Membership Pattern** is used next, to introduce the *memberOf* relation between the *Potential Car Renter Community* subkind and the *Potential Car Renter* mixin. Finally, to complete the model we apply the **Relator Pattern** Variant 1 ($k = 2$) twice, to introduce the two relators *Service Offering* and *Car Rental Offering*, which are then connected using the **Inheritance Pattern**. After this step, all relational dependencies are satisfied, and are removed via two applications of the **Relational Dependency Pattern** Variant 1 (handling RDP-1 and RDP-2), and two applications of the **Relational Dependency Pattern** Variant 2 (handling RDP-3 and RDP-4). This concludes the model creation, yielding the graph shown in Fig. 8.

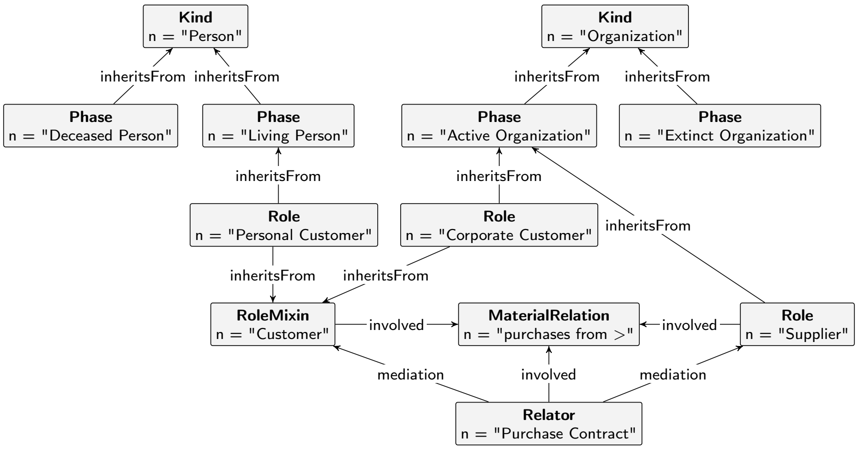


Fig. 7. Model from [5] produced in GROOVE using the proposed Ontology Pattern Grammar.

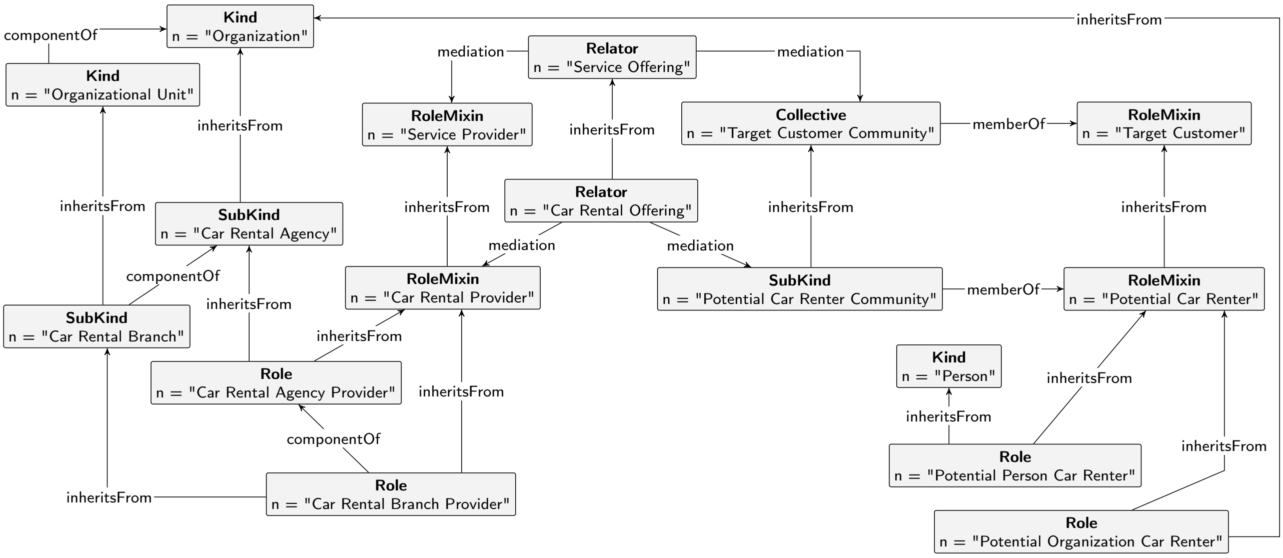


Fig. 8. Model from [5] produced in GROOVE using the proposed Ontology Pattern Grammar.

VII. Conclusion

In this paper, we employ the formalism of a graph grammar to propose an alternative formulation of the OntoUML language, fully defining it as an Ontology Pattern Grammar. Given that (as shown in [4]) OntoUML is among the most used modeling languages for ontology-driven conceptual modeling, we believe that the results presented here amount to a theoretical and practical contribution to the conceptual modeling community.

In an extended version of this paper, we shall elaborate on a version of the OntoUML modeling tool that fully implements a computational support for the modeling strategy proposed here, in which models are completely constructed by the restricted combination of these patterns as higher-granularity modeling primitives (see discussion in [5]). We believe that this strategy dramatically reduces the complexity of the modeling process, especially for novice modelers. This is, of course, an empirical question, which we intend to address in a series of experiments.

As discussed in [6], the observation of the application of OntoUML over the years conducted by a variety of groups in a variety of domains amounted to a fruitful empirical source of knowledge that triggered the evolution of both UFO and OntoUML. In this process, termed *Systematic Subversions* [6], users of the language systematically created models that were (purposefully) grammatically incorrect but which were needed to express the intended characterization of their underlying conceptualizations that could not be expressed otherwise. These includes the representation of event-related phenomena in structural conceptual models [17], the representation of *powertypes* [18] (types whose instances are types, not individuals), as well as the representation of anti-rigid types (*e.g.*, roles, phases) and non-sortal types (*i.e.*, categories, mixins and role-mixins, whose instances are moments (relators, modes) [19]. As a follow up of this paper, we intend to propose an updated version of the OntoUML pattern grammar presented here to formally account for new patterns and constraints related to the modeling of these phenomena.

We would like to highlight that the definition and implementation of the language as presented here bring several benefits to this community, namely: (i) the grammar is defined in a formal, Turing powerful, computational method that circumvents the limitations of the current meta-modeling approaches for defining the abstract syntax of modeling languages; (ii) the language is defined in a way that affords its independence from the UML meta-model and, as consequence, the results presented here can be ported to other conceptual modeling languages and employed by the conceptual modeling community at large; and (iii) the language makes explicit its constituting ontology design patterns which, once more, reflect the ontological micro-theories put forth by UFO. In other words, in comparison to the current definition of OntoUML's abstract syntax (in terms of a UML 2.0 meta-model with associated OCL constraints), the implementation of this language in the manner proposed here affords a much higher *ontological transparency* for the language, *i.e.*, the implementation makes much more transparent the ontological commitments embedded in that conceptual modeling language.

In summary, we believe to have proposed in this paper what is (to the extent of our knowledge) the first attempt to produce a general-purpose conceptual modeling language that is ontologically well-founded, explicitly defined as an ontology pattern language, and not tied to a particular legacy meta-model (the UML 2.0 meta-model).

REFERENCES

[1] J. Recker, M. Rosemann, P. Green, and M. Indulska, " Do ontological deficiencies in modeling grammars matter?" *MIS Quarterly*, vol. 35, no. 1, pp. 57–79, 2011.

[2] G. Guizzardi and G. Wagner, " Using the Unified Foundational Ontology (UFO) as a oundation or general conceptual modeling languages," *Theory and Applications of Ontology: Computer Applications*, pp. 175–196, 2010.

[3] G. Guizzardi, *Ontological oundations or structural conceptual models*, ser. Telematica Institute Fundamental Research Series. University of Twente, 2005, no. 15.

[4] M. Verdonck and F. Gailly, " Insights on the use and application of ontology and conceptual modeling languages in ontology-driven conceptual modeling," *ER (LNCS)*, pp. 83–97, 2016.

[5] F. Ruy, G. Guizzardi, R. Falbo, C. Reginato, and V. Santos, " From reference ontologies to ontology patterns and back," *Data & Knowledge Engineering*, 2017.

[6] G. Guizzardi, G. Wagner, J. Almeida, and R. Guizzardi, " Towards ontological oundation or conceptual modeling: the Unified Foundational Ontology (UFO) story," *Applied Ontology*, pp. 259–271, 2015.

[7] A. Rensink, " The GROOVE Simulator: A tool or state space generation," *AGTIVE (LNCS)*, pp. 479–485, 2003.

[8] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova, " Modelling and analysis using GROOVE," *STTT*, vol. 14, no. 1, pp. 15–40, 2012.

[9] T. Halpin, " Object-role modeling: principles and benefits," *Int. J. Inf. Syst. Model. Des.*, vol. 1, no. 1, pp. 33–57, 2010.

[10] T. Halpin and T. Morgan, *Information modeling and relational databases*, 2nd ed. Morgan Kaufmann, 2008.

[11] R. Heckel, " Graph transformation in a nutshell," *ENTCS*, vol. 148, no. 1, pp. 187–198, 2006.

[12] E. Zambon, *Abstract Graph Transformation – Theory and Practice*, ser. Centre or Telematics and Information Technology. University of Twente, 2013.

[13] A. Habel and D. Plump, " Computational completeness of programming languages based on graph transformation," *FoSSaCS (LNCS)*, pp. 230–245, 2001.

[14] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, " Attributed graph transformation with node type inheritance," *Theor. Comput. Sci.*, vol. 376, no. 3, pp. 139–163, 2007.

[15] R. Grønmo, S. Krogdahl, and B. Møller-Pedersen, " A collection operator or graph transformation," *ICMT (LNCS)*, pp. 67–82, 2009.

[16] A. Rensink and J.-H. Kuperus, " Repotting the geraniums: On nested graph transformation rules," *GT-VMT*, 2009.

[17] G. Guizzardi, J. Almeida, and N. Guarino, " Ontological Considerations About the Representation of Events and Endurants in Business Models," *BPM*, pp. 20–36, 2016.

[18] G. Guizzardi, J. Almeida, N. Guarino, and V. Carvalho, " Towards an Ontological Analysis of Powertypes," *IJCAI*, 2015.

[19] N. Guarino and G. Guizzardi, " We Need to Discuss the Relationship: Revisiting Relationships as Modeling Constructs," *CAiSE (LNCS)*, 2015.